KASSPER Real-time Embedded Signal Processor Testbed

Glenn Schrader, Andrew Heckerling, and Michael Harrison

***Abstract*—This paper describes the work that is being done by MIT Lincoln Laboratory (MIT LL) to develop a real-time signal processing testbed for DARPA's Knowledge-Aided Sensor Signal Processing and Expert Reasoning (KASSPER) project.**

***Index Terms*—KASSPER, real-time testbed, signal processor, implementation**

I. INTRODUCTION

THIS paper describes the work which is being done by MIT Lincoln Laboratory (MIT LL) to develop a real-time signal processing testbed for the Knowledge Aided Sensor Signal Processing and Expert Reasoning (KASSPER) project. The goal of the KASSPER project is to improve the performance of Ground Moving Target Indicator (GMTI) radar systems by making use of a-priori data and/or expert reasoning techniques. The first section of this paper will describe the signal processing testbed itself. The second part of the paper will describe the Parallel Vector Library (PVL), which is a parallel signal processing middleware that as been developed at MIT LL and will be as part of the testbed's software infrastructure.

II. THE KASSPER SIGNAL PROCESSING TESTBED

The goal of the real-time testbed is to provide a means to explore the implementation issues related to KASSPER processing. In order to achieve this, the design of the system will be driven by a few key system level concepts. These system level concepts will drive the selection of the real-time testbed's software architecture. The system level concepts and software architecture are intentionally generic with respect to any real-time signal processor that could be used. A brief description of the signal processor that will be used in the real-time testbed will also be provided.

A. System Level Concepts

Look ahead scheduling, KASSPER algorithm implementation, and intelligent caching are areas that MIT LL will be investigating. In this section, we will discuss each of these concepts

The concept behind "look ahead scheduling" is that the radar system needs to combine knowledge of the goals of the system's current mission with knowledge about the external environment that the system finds itself in order to pre-schedule the system's processing timeline some amount of time into the future. For instance, the system may be able to determine that there are optimum directions from which to view an area of interest. The system can then schedule the viewing of different regions such that each is viewed from the "best" direction.

In order to effectively implement KASSPER signal processing algorithms, the computational requirements, data transfer bandwidth requirements, and the amount of inherent parallelism for each algorithm will need to be assessed. For instance, KASSPER signal processing algorithms will use environmental a-priori knowledge to augment their processing of radar returns. The amount of potential a-priori data (i.e., Digital Terrain Elevation Data (DTED) databases, etc.), a very large amount of additional processing and/or data movement may also be necessary. Both the computational and data handling characteristics of potential KASSPER algorithms need to be studied in order to determine the best implementation approach.

KASSPER processing that depends on a-priori knowledge will rely on prompt real-time access to that knowledge when the processing is performed. The potentially very large volume of a-priori data that must be available will require that the data will be stored on magnetic disks. Since the latency of accessing such a disk based database can be large, it will be necessary to intelligently pre-load a knowledge cache with the data that the processor will need in the near future.

B. Processor Architecture

Two key observations that result from the above system level concepts are that a real-time knowledge data will be required and that multiple signal processing algorithms (i.e., multiple processor modes) must be supported. This section will provide a high level description of these aspects of the processor architecture.

A block diagram of the knowledge database architecture is shown in Fig. 1. The knowledge database provides storage for a-priori information that can be used as input to KASSPER processing algorithms. The amount of a-priori data can be very large and it is assumed that secondary mass storage such as hard disks will be necessary. From a real-time point of view the long latency that is incurred when accessing data from a hard disk is unacceptable. The a-priori data must therefore be pre-loaded into a cache by the processor in advance of the data actually being used. Since the radar's processing is being scheduled into the future via look ahead scheduling, the cache mechanism can use the projected schedule along with the platform's environment data to select the appropriate knowledge data to be loaded and/or dropped out of the knowledge cache. Note that the scheduler must keep a running index of the current location of all knowledge data so that it can manage the contents of the knowledge cache.

Note that knowledge data typically cannot be directly used as it comes out of the knowledge database. For instance, geographic coordinates (longitude, latitude) need to be transformed into radar system coordinates (beam, range cell) before they can be used. Some amount of pre-processing will almost certainly be needed for all data retrieved from the knowledge database.

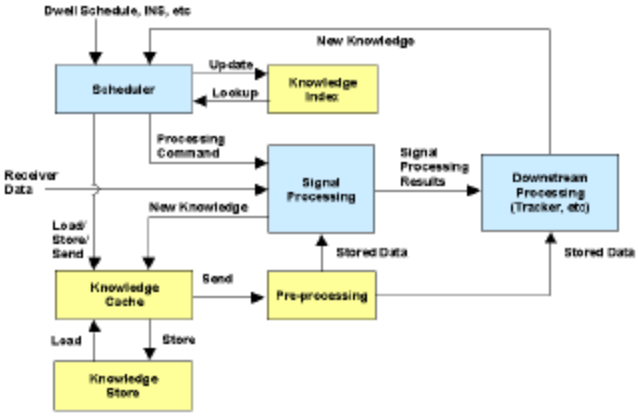

Fig. 1. Knowledge Database Architecture Block Diagram

A block diagram of the testbed's mult-mode architecture is shown in Fig. 2. Due to the complexity of the "real world" environment, it is expected that both traditional GMTI, Synthetic Aperture Radar (SAR), as well as GMTI/SAR variants might be necessary to achieve the desired radar detection performance. The system must therefore support multiple processing algorithms and allow switching between them at runtime based on the current conditions. This can thought of as a set of independent processing pipelines where each pipeline implements a different algorithm. Input data can be directed to the appropriate algorithm pipeline and the results gathered from the output of that pipeline.

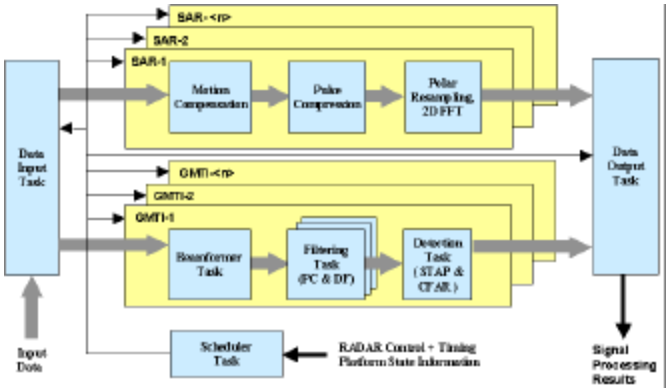

Fig. 2. Multi-mode Architecture Block Diagram

Note that these pipelines should be thought of as logical pipelines rather than physical pipelines. When these algorithms are implemented on a real-time signal processor, they will simply be different processing modes within the application software.

In order to prototype the multi-mode architecture, the baseline GMTI algorithm shown in Fig. 3 and the baseline SAR algorithm shown in Fig. 4 have been defined. These algorithms are traditional GMTI and SAR algorithms that make no use of KASSPER concepts. However, the overall processor architecture will support the inclusion of knowledge so these implementations can be used as a basis for implementing other algorithms which do use KASSPER concepts.

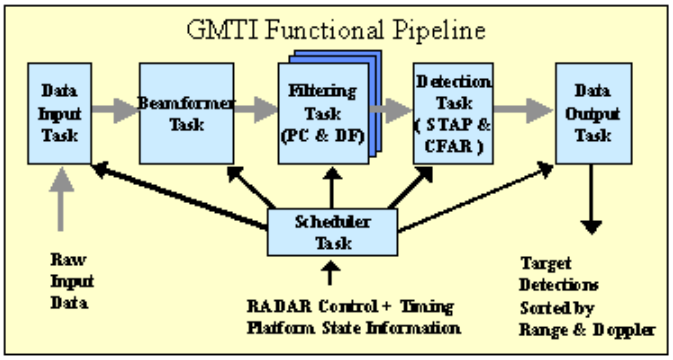

Fig. 3. Baseline GMTI Algorithm Block Diagram

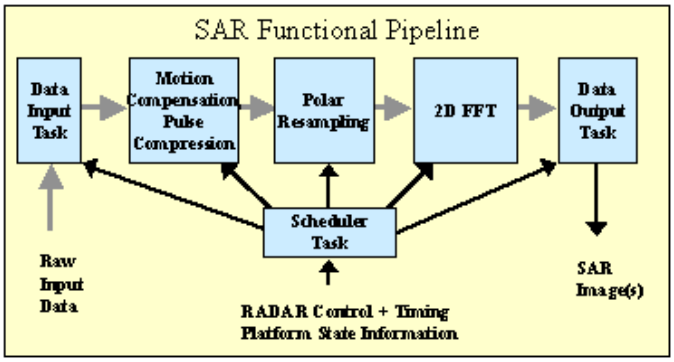

Fig. 4. Baseline SAR Algorithm Block Diagram

C. Processor Hardware Architecture

The testbed signal processor is based on a Mercury Computer Systems parallel multi-processor. Note that this processor is typical of the real-time processors that are used to implement radar signal processors and is therefore a reasonable system on which to investigate KASSPER algorithm implementation issues. The Mercury processor, when fully expanded, will have 120 Power PC G4 processors. This will provide an aggregate peak computational throughput of approximately 480 Gflop/sec. This processor is based on Mercury's Power Stream architecture, which delivers approximately 8.5 Gbyte/second of communication bandwidth (bisection bandwidth).

One of the goals of the testbed hardware environment is to provide interfaces to the signal processor that are similar to the interfaces that could be encountered on an actual platform. In the event this processor is integrated onto an actual platform this approach will allow many of the real-time timing and control issues to be worked out prior to attempting to integrate the processor with the platform.

III. The Development Process

The software development team at MIT LL follows the process that is diagrammed in Fig. 5. The four major steps in this process are algorithm design, algorithm verification, software design, and software implementation.

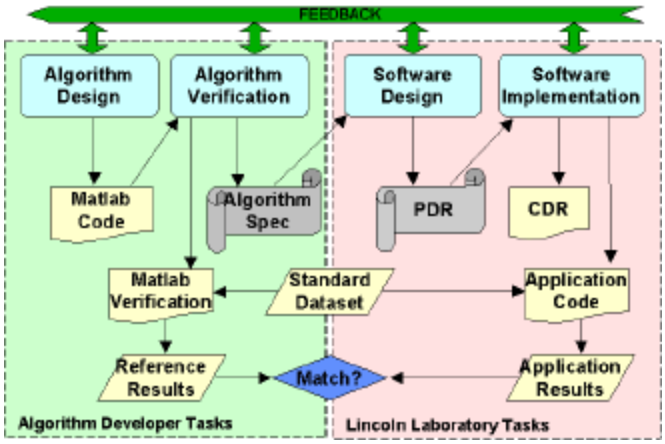

Fig. 5. Real-time Testbed Software Development Process

An algorithm team will perform the algorithm design and verification. The algorithm team will then produce a Matlab reference implementation of the algorithm and an algorithm specification document. This algorithm team could be either a MIT LL team or a team within some other KASSPER contributor. The MIT LL software development team will perform the software design and implementation. The software development team will produce a Preliminary Design Review document (PDR, a high level software design) and the actual application code. This process has proven to enable both higher software productivity and, therefore, quicker algorithm insertion.

IV. The Parallel Vector Library

A. The Motivation for Creating PVL

Developing high-performance parallel signal processing software has traditionally been very difficult. Making a parallel application both portable and scalable adds even more difficulty. Also, rapid prototyping is difficult when developing directly on a real-time processor. It has long been realized in the scientific computing community that an appropriate computational middleware can greatly ease parallel software development. However, the type of problems which scientific programming has traditionally addressed (i.e., solution of a single large set of linear equations) is not a good match for the structure of many signal-processing problems (i.e., processing pipelines which compute a large number of relatively small problems on streaming data). PVL leverages a number of the techniques developed for scientific parallel computing to produce a middleware that is more suited to parallel signal processing.

B. Parallel Signal Processing Application Concepts

Many signal processing problems can be visualized by a signal flow graph (i.e., boxes which represent data transformations connected by arrows representing data movement). PVL represents the computation blocks as objects called Tasks and the communication flows as objects called Conduits. Fig. 6 shows a simple application consisting of tasks that supply data, perform computation, and consume results.

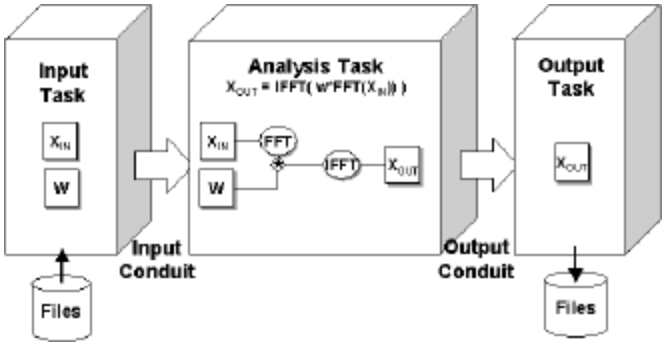

Fig. 6. Simple Pipelined Application

Note that each Task contains parallel data objects and/or parallel computation objects. All of these objects (including Tasks) are 'mappable'. The mapping of a parallel object is a description of the way in which that object is distributed across the computational elements of the parallel processor.

C. Application Development and Mapping

A common approach to developing parallel software is to write a single application that is run across all of the processors. This is known as the Single Program Multiple Data (SPMD) model. Note that each processor must decide which portion of the application's processing it will perform based upon the application's mapping. A traditional way of writing such

applications is to directly code this decision logic into the application. While this can and has worked in the past, the application becomes brittle with respect to scaling. For instance, if the number of processors needs to be changed then modifications to the application source code become necessary. It also becomes necessary to make coordinated changes in portions of the application that are inter-related. This makes code maintainability difficult and costly. Moving this decision logic into a middleware library is a significant improvement since the logic is concentrated in the library rather than scattered throughout the application. Since this a common problem across many applications, a middleware solution also avoids re-implementing the functionality for each application.

The work performed by MIT LL on previous parallel processor efforts resulted in a number of observations and insights that influenced the design of PVL:

- Algorithm implementation and mapping of the application to the signal processor should be as orthogonal as possible. Changing the scaling (i.e., number of processors) or the processor CPU architecture can happen both during development and as technology upgrades are added in the field. Application re-writes as a result of this are expensive both during development and during system maintenance.
- Computation and communication should be seamless from the application's point of view. The more that the parallel architecture of the runtime system becomes visible to the application the more changes in scaling and architecture affect the application code. Ideally, communication should be performed implicitly as needed based on the mapping of the data objects and the operations being performed on them.
- The combination of the previous two points makes rapid prototyping easier since application code can be portable between workstation and real-time environments. Workstations or a network of workstations (NOW) have excellent debugging tools, are readily available, and are relatively inexpensive. Real-time parallel processors tend to have less mature debugging tools, are expensive and therefore scarce. Developing a functionally correct application in a workstation or NOW environment and then moving the application onto the real-time parallel processor environment to address real-time issues results in an overall development methodology which leverages the best strengths of each available tool.
- Much of the code in an application does not affect the actual performance of the application (i.e., setup or error handling code). Approximately 90% of the signal processing workload comes from approximately 10% of the code. Object oriented techniques can be used to make the 90% of the code that is not critical to performance as easy to develop as possible.
- There will always be some application specific computational kernels that a general-purpose library such as PVL will not support. The library must therefore supply data access 'hooks' which allow such computational kernels to be built by the application programmer in an efficient manner.

D. A Simple PVL Example

The following very simple code example highlights several key PVL attributes.

```
1.  #include <Vector.h>
2.  #include <AddPvl.h>
3.  void addVectors(aMap, bMap, cMap) {
4.      Vector< Complex<Float> > a('a', aMap, LENGTH);
5.      Vector< Complex<Float> > b('b', bMap, LENGTH);
6.      Vector< Complex<Float> > c('c', cMap, LENGTH);
7.  a = 0;  b = 1;  c = 2;
8.  a=b+c;
9.  }
```

Vector (see lines 4,5,6) is a class that provides an abstraction of a mathematical vector. The Complex<Float> between brackets is a template parameter indicating that the elements in the vector are complex and that the real and imaginary parts are each of type Float. Note that a map is supplied to create each Vector. The operations on lines 7 (filling the vectors with a value) and line 8 (placing the element-wise sum of two vectors into a third vector) remain the same regardless of the maps supplied to the vectors. If any communications is necessary to perform the operations, then it happens 'under the covers' and is of no concern (functionally) to the application programmer.

E. The structure of PVL

Since PVL is a middleware, it provides services to an application program via an Application Programming Interface (API) and makes use of the services of other supporting libraries via their APIs. PVL is typically built to use the Vector, Signal, and Image Processing Library (VSIPL) to perform computation and the Message Passing Interface (MPI) to perform communication. Both of these are open standard APIs. PVL itself, and therefore PVL applications, are very portable between platforms that support VSIPL and MPI.

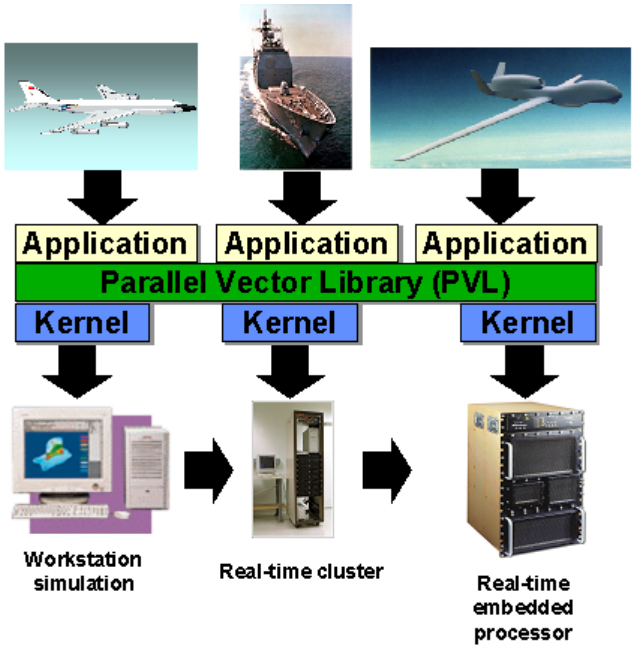

Fig. 7. PVL Architecture

F. Future Directions for PVL

There are two additional efforts that have grown out of MIT LL's work on PVL.

Hardware PVL (HPVL) extends PVL by including support for computational resources such as Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs). In some cases these may be a more appropriate choice for implementing an algorithm than a general purpose. The goal of HPVL is to develop an architecture that will allow processing to be mapped to either software based computational kernels or hardware based computational kernels.

Parallel Matlab provides PVL-like parallel constructs within Matlab programs. A unique feature of Parallel Matlab is that the middleware is implemented entirely in Matlab code. Parallel Matlab programs should therefore be portable to any platform that is able to run Matlab.

V. SUMMARY

MIT LL has made significant progress defining system level concepts as well as the software and hardware architectures that will be used in the KASSPER real-time testbed. Over the next year we will begin developing prototype implementations these architectures on the Mercury parallel processor. In addition, we will continue our ongoing work on implementing our baseline processing algorithms.